
CEWES MSRC/PET TR/99-25

# Improving Performance by Scalar Replacement in a CFD Code

by

M. Ehtesham Hayder
John Mellor-Crummey

05h00199

**Work funded by the DoD High Performance Computing Modernization Program CEWES Major Shared Resource Center through**

Programming Environment and Training (PET)

Supported by Contract Number: DAHC94-96-C0002
Nichols Research Corporation

# Improving Performance by Scalar Replacement in a CFD Code

M. Ehtesham Hayder and John Mellor-Crummey
Center for Research on Parallel Computation
Rice University, Houston, TX 77005-1892

## Abstract

We consider a moderate size computational fluid dynamics application code known as HELIX for a study of improving performance by scalar replacement, i.e., loading array element values that are used repeatedly into scalars to enable them to be register allocated by the back-end compiler. The goal of scalar replacement is to reduce load/store traffic. Our experiments compare three versions of HELIX – the original code, a version in which scalar replacement was performed by hand, and version in which scalar replacement was performed automatically by Memoria – a tool for performing scalar replacement developed at Rice University and Michigan Technological University. Our experiments show that scalar replacement improves performance of this code by 4 to 12% over and above performance obtained by using the highest level of optimization with vendor-supplied compilers on an SGI Origin, an SGI O2 workstation, an IBM SP, and a CRAY T3E. Improvements in the execution time were primarily due to the reduction in the number of load/store instructions.

## 1. Introduction

Architectures of modern supercomputers are continuously changing. An optimal code for traditional vector machines is not optimal for newer machines that are based on deep memory hierarchies. A developer of an application has knowledge of the algorithm and dependences among variables and sections of the code. With some knowledge of optimization techniques, he may be able to improve the performance of the application. Compilers on the other hand have to discover all optimization opportunities from the code, which at times may be difficult, if not impossible. Sometimes compiler based program transformations can aid the developer in achieving high performance by restructuring the code at the source level so that a back-end compiler can generate efficient code. In this study we concentrate on a particular type of optimization, namely scalar replacement. Motivation for our present work comes from the code migration group at CEWES MSRC, who indicated that scalar replacement could be used to significantly improve performance of some of their codes. We undertook this study to examine a CFD code of interest to CEWES MSRC. We performed our experiments on a SGI O2K, an SGI O2 workstation, CRAY T3E and an IBM SP. All computations were performed on a single node, since single process optimization is the focus of this study.

We present brief descriptions of the code in the next section, followed by a description of our optimization technique. Simulation results and conclusions are presented in following sections.

## 2. Application

The application code we consider in this study is known as the HELIX code [Ramachandran et al., 1989; Steinhoff and Ramachandran, 1990]. This code is used for flow computation of interest to helicopter designers. It solves the unsteady full potential equation in three dimensions on an Eulerian grid with embedded vortical velocity that models the influence of wake. Such results are used to calculate vibratory air loads in forward flight and performance in hover. Computations are done to solve

$$\nabla.(\rho V) = 0$$

where $\rho$ is the density and $V$ is the velocity. The velocity is decomposed into rotational ($q^v$) and irrotational ($\nabla\phi$) parts. In addition there is also contribution from rotational coordinate transformation. An iterative technique is used to obtain the flow field. In brief, it has following four steps:

1. The vortex sheet position is integrated as a set of marker streamlines which follow the flow using interpolated values of V from the fixed grid.
2. Rotational velocity contribution ($q^v$) is computed at grid points near the sheet.
3. A potential ($\phi$) is computed at all grid points by solving the compressible mass conservation equation.
4. A new velocity V is computed at each grid point after adding the rotational velocity contribution to the potential and free stream components of the velocity. At convergence the vortex sheet follows the flow.

HELIX is a moderate size FORTRAN code with about 6500 source lines. The most compute intensive section of the code corresponds to item 3 above. This computation is performed by the subroutine YSWEEP. A relaxation technique is used for each radial (K) plane, i.e. from upstream to downstream in the radial, and from upper axial boundary to bottom axial boundary in the azimuthal direction. Loop 32 within the YSWEEP routine computes metric terms of computational cells, derivative terms for the potential ($\phi$) and velocities for interior cells. Subroutine MIXFLO is a central program which directs the flow of the code by calling YSWEEP for each radial (K) plane and setting up marker coordinates for next free wake calculation. MIXFLO is called by the main program.

## 3. Optimization

### 3.1 Techniques

There are various optimization techniques which one may use to improve performance of a code. Our present study focuses on a specific type of optimization, namely scalar replacement. We provide a brief introduction to this optimization. Other types of optimizations, such as elimination of common sub expressions, loop unrolling, unit stride access, etc. may also improve the performance of a particular code significantly. Amount of improvements to a code will depend on the algorithm and how it is implemented.

In modern computers there are different types of storage with different access times. If a variable resides in the register, it can be accessed in a clock cycle without generating memory traffic. When an array is accessed in some regular pattern, say in a typical finite difference computation, some variables used for computations at a grid point may be may be reused in computations at neighboring grid points. In many cases, load store traffic can be reduced for some of these references by exploiting temporal locality and storing appropriate variables in registers. One way to hint to a back-end compiler that an array reference should be stored in a register to reduce load/store traffic by exploiting temporal reuse is to replace references to arrays with scalars. In the following code

```
DO  I=2, N
 A(I) = C1*A(I) + C2*(A(I+1) - A(I-1))
ENDDO
```

A(I-1) is generated in the in the previous iteration, which may be saved in a register for reuse by the next iteration without loading the value from memory. Such optimization opportunities should be discovered by a good compiler. However, some compilers may miss such opportunities. For these compiler some restructuring of the code as shown below will make it easier to generate efficient executables.

```
T = A(1)
DO  I=2, N
 T =  C1*A(I) + C2*(A(I+1) - T)
 A(I) = T
ENDDO
```

Additional discussion on scalar replacement may be found elsewhere (see Carr and Kennedy, 1994a and 1994b; Callahan et al., 1998). There are also many other optimization techniques, such as loop nest optimization, loop unrolling, etc. which a code developer may use to make the job of a compiler easier. Discussion on these techniques may also be found in the above reference.

### 3.1 Tools

Some shortcomings in vendor compilers may be overcome by using a preprocessor which will modify the code so that the compiler will be able to easily find temporal localities. One such tool is Memoria which uses optimization techniques discussed by Carr and Kennedy, 1994a and 1994b to generate code with optimal scalar replacement. This tool was initially developed at Rice University and has been further enhanced at Michigan Technological University by Steve Carr and his coworkers. Input to Memoria consists of a Fortran source code along with information about the target machine architecture. Memoria performs dependence analysis to generate an improved code. Since scalar replacement is already done in the code, the compiler does not need to worry about this particular optimization.

### 3.2 Modifications to the code

We experimented with three versions of the HELIX code in this work. They are

- **The original code:** This version of the code was given to us by the code migration group at CEWES MSRC. It is written with many array references.
- **Hand tuned code:** We performed scalar replacement in two key loops (Do loops 32 and 105) in YSWEEP routine by hand. Loop 32 has been described in Section 2. Computations in loop 105 are similar to those in loop 32, except it computes at the boundaries whereas computations in loop 32 are for interior points in the domain
- **Memoria modified code:** Memoria performed scalar replacements in the YSWEEP routine of the original code. Other routines in HELIX were not modified. We limited scalar replacement only to YSWEEP to make a roughly fair comparison between this (Memoria modified version) and the hand tuned version.

In the following we give portions of loop 32 in our three versions:

```
      DO 15   I = 1, NX+1
      X(I)      = X3D(I,J,K)
      XM(I)     = X3D(I,J+1,K)
      XR(I)     = X3D(I,J,K+1)
      XRM(I)    = X3D(I,J+1,K+1)
      ******
      ******

15    CONTINUE
```

```
C
      DO 32 I=1,NX
         XX        = XR(I+1)  -XR(I)  +XRM(I+1)  -XRM(I)
         XXS       = +X(I+1)  -X(I)  +XM(I+1)  -XM(I)
         XX        = XX +XXS
         XY        = XR(I+1)  +XR(I)  -XRM(I+1)  -XRM(I)
         XYS       = +X(I+1)  +X(I)  -XM(I+1)  -XM(I)
         XY        = XY +XYS
         XZ        = XR(I+1)  +XR(I)  +XRM(I+1)  +XRM(I)
     &        -(X(I+1)  +X(I)  +XM(I+1)  +XM(I))
         *******
         XAV       = (X(I)+X(I+1)+XM(I)+XM(I+1)+
     &        XR(I) +XR(I+1) +XRM(I) +XRM(I+1))*0.125
         *******
         *******
32       continue
```

**Figure 3.1: Part of original loops 15 and 32**

```
         DO 32 I=1,NX
          XSUBI     = X3D(I,J,K)
          XSUBIP1   = X3D(I+1,J,K)
          XMSUBI    = X3D(I,J+1,K)
          XMSUBIP1  = X3D(I+1,J+1,K)
          XRSUBI    = X3D(I,J,K+1)
          XRSUBIP1  = X3D(I+1,J,K+1)
          XRMSUBI   = X3D(I,J+1,K+1)
          XRMSUBIP1 = X3D(I+1,J+1,K+1)
          XX        = XRSUBIP1 -XRSUBI +XRMSUBIP1 -XRMSUBI
     &          +(XSUBIP1  -XSUBI  +XMSUBIP1  -XMSUBI)
          XY        = XRSUBIP1 +XRSUBI -XRMSUBIP1 -XRMSUBI
     &          +(XSUBIP1 +XSUBI  -XMSUBIP1  -XMSUBI)
          XZ        = XRSUBIP1 +XRSUBI +XRMSUBIP1 +XRMSUBI
     &        -(XSUBIP1 +XSUBI +XMSUBIP1 +XMSUBI)
          XAV       = (XSUBI +XSUBIP1 +XMSUBI +XMSUBIP1
     &        + XRSUBI +XRSUBIP1 +XRMSUBI +XRMSUBIP1)*.125
```

```
         *******
         *******
32       continue
```

**Figure 3.2: Hand modified code of section shown in Figure 3.1**

```
        do i = 1, nx + 1
          x(i) = x3d(i, j, k)
          xm(i) = x3d(i, j + 1, k)
          xr(i) = x3d(i, j, k + 1)
          xrm(i) = x3d(i, j + 1, k + 1)
         ******
        enddo

        do i = 1, nx
          xr$0$0 = xr(i + 1)
          xr$1$0 = xr(i)
          xrm$2$0 = xrm(i + 1)
          xrm$3$0 = xrm(i)
          xx = xr$0$0 - xr$1$0 + xrm$2$0 - xrm$3$0
          x$4$0 = x(i + 1)
          x$5$0 = x(i)
          xm$6$0 = xm(i + 1)
          xm$7$0 = xm(i)
          xxs = x$4$0 - x$5$0 + xm$6$0 - xm$7$0
          xx = xx + xxs
          xy = xr$0$0 + xr$1$0 - xrm$2$0 - xrm$3$0
          xys = x$4$0 + x$5$0 - xm$6$0 - xm$7$0
          xy = xy + xys
          xz = xr$0$0 + xr$1$0 + xrm$2$0 + xrm$3$0 - (x$4$0 + x$5$0 + xm
     *$6$0 + xm$7$0)
          *******
          xav = (x$5$0 + x$4$0 + xm$7$0 + xm$6$0 + xr$1$0 + xr$0$0 + xrm
     *$3$0 + xrm$2$0) * 0.125
          *******
          *******
          enddo
```

**Figure 3.3: Memoria modified code of section shown in Figure 3.1**

## 4. Results

Results of our study with the HELIX code computations on 101x59x29 grid using a single SGI O2K node are shown in Table 4.1. Profiling the original code shows that most (over 90%) of the computation time is spent in the YSWEEP routine. We focused our tuning efforts on to two key loops (loop 32 and loop 105) in this routine. Memoria preprocessing was limited to the YSWEEP routine to get a reasonable comparison between hand tuned code and Memoria generated code. We used SGI's Perfex tool (see Speedshop user's guide for description of this utility) on the SGI O2K and the O2 workstation to gather performance statistics. Tests on the SGI O2K were done by running the job in the batch mode while other users were also sharing the machine. The O2 workstation was not shared with any other user for our measurements. All statistics were gathered multiple times and we report the averages. Measurements which showed noticeable variations from the mean were discarded. We used -O0 (no optimization) and -O3 (aggressive optimization) compiler options to generate executables. In Table 4.1a we show execution time for 10 iterations, % of primary and secondary data cache misses and TLB misses. We chose to limit our tests to only 10 iterations to keep computation times reasonable while maintaining a fairly good representation of computational characteristics. The hand tuned version is about 10% faster than the original version for no optimization case. However, the gain dropped to about 4% for aggressive optimization case. Memoria generated code reduced computation time by about 3.5% for no optimization case, but the original version performed slightly better with aggressive optimization. When aggressive compiler option (-O3) is used, the compiler goes through detailed analysis of the code and performs a significant amount of optimization which we performed on the source code by hand and Memoria performed automatically. Nevertheless, hand tuned versions performed better than the original version of the code for all compiler options. There are less than 1% primary data cache misses when no optimization (-O0) flag is turned on, while it increased about ten fold to approximately 8% for -O3 option. The primary data cache hit ratio is one measure of optimization, but it is not the only measure. For example, say a variable $x$ is stored in a register instead of keeping it in cache and the program accesses $x$ repeatedly. Then there will be only one cache miss initially and $x$ will be loaded to a register. Since $x$ now resides in a register, it will be used without any need to access it from cache, i.e., no cache hit. On the other hand, if $x$ was not loaded on a register, there would be a cache hit for subsequent uses of $x$. If all other data access patterns remain unchanged, then the program with $x$ assigned to a register will run faster although it will have higher percentage of cache misses. In any case, we believe a 8% cache miss ratio is high and this high percentage of primary data cache misses with the -O3 option is related to the data access pattern or the structure of the code. A close examination of loop 32 shows that many variables are accessed along a line while their structures are three- dimensional. Many data elements (field variables) are accessed repeatedly and used only once in different phases of computations. Due to the limited size of primary cache, these elements are evicted from cache between their uses. One may be able to restructure the code without changing the algorithm so that a sweep is performed along a column instead of a pencil in the three-dimensional field. One may also be able to design the iterative algorithm to ensure better

temporal locality of data.

Our primary focus in this study is optimization using scalar replacement, which basically reduces load/store traffic. We examine this traffic for our three versions of the code in Table 4.1b. In this table we compare execution times, number of graduated loads and stores with original code compiled with -O0 option (i.e. no optimization) considered as the base case. When -O0 flag was used, the hand tuned and Memoria generated versions had 19% and 2% respectively, fewer number of loads than the original version. The number of stores increased for both versions, and it went up by as much as 20% for Memoria generated version. For aggressive optimization, the hand tuned version reduced load and store by 7% and 14% respectively, over the original code. In the Memoria generated version, stores decreased by 12% while loads increased by 6%.

| Version | OPT | time (sec) | % pr. cache miss | % sec cache miss | % tlb miss |
|---|---|---|---|---|---|
| Original | O0 | 6326.263 | 0.7861 | 3.8076 | .0012 |
| Original | O3 | 1226.373 | 8.1450 | 3.0209 | .0046 |
| Hand tuned | O0 | 5670.373 | 0.8147 | 4.4210 | .0013 |
| Hand tuned | O3 | 1175.877 | 8.0955 | 3.2951 | .0052 |
| Memoria generated | O0 | 6107.141 | 0.7877 | 3.7964 | .0012 |
| Memoria generated | O3 | 1242.653 | 7.9090 | 3.0641 | .0039 |

**Table 4.1a: Execution time and data misses**

| Version | OPT | time | load | store |
|---|---|---|---|---|
| Original | O0 | 1 | 1 | 1 |
| Original | O3 | 0.1939 | 0.1575 | 0.5249 |
| Hand tuned | O0 | 0.8963 | 0.8140 | 1.0419 |
| Hand tuned | O3 | 0.1859 | 0.1469 | 0.4517 |
| Memoria generated | O0 | 0.9654 | 0.9815 | 1.200 |
| Memoria generated | O3 | 0.1964 | 0.1673 | 0.4635 |

**Table 4.1b: Comparisons with the base case**

**Table 4.1: Computation of HELIX on a single node O2K**

We repeated our tests on an O2 workstation. Both of these machines use the MIPS R10000 processor. Test on the workstation were done because of our easy access to this machine and also they provide additional data on a similar platform for verification of data in our experiments. Results on the O2 workstations are shown in Table 4.2 and they are in the same format as in Table 4.1. These results follow the same general trend except secondary data cache and TLB misses on the O2 workstation are much higher than those on the O2K node. This was due to smaller size L2 cache and page size on the O2 workstation. Cache sizes for O2K and O2 workstation are given in Table 4.3

| Version | OPT | time (sec) | % pr. cache miss | % sec cache miss | % tlb miss |
|---|---|---|---|---|---|
| Original | O0 | 7603.6577 | 0.8565 | 11.0561 | .0182 |
| Original | O3 | 1497.8949 | 6.1064 | 9.5015 | .0804 |
| Hand tuned | O0 | 6712.4883 | 0.8524 | 11.5299 | .0176 |
| Hand tuned | O3 | 1420.8648 | 5.9182 | 10.4348 | .0707 |
| Memoria generated | O0 | 7287.0278 | 0.9348 | 10.6817 | .0196 |
| Memoria generated | O3 | 1500.3543 | 5.9531 | 9.5009 | .0791 |

**Table 4.2a: Execution time and data misses**

| Version | OPT | time | load | store |
|---|---|---|---|---|
| Original | O0 | 1 | 1 | 1 |
| Original | O3 | 0.1970 | 0.0869 | 0.4706 |
| Hand tuned | O0 | 0.8828 | 0.8553 | 1.0345 |
| Hand tuned | O3 | 0.1869 | 0.0845 | 0.3941 |
| Memoria generated | O0 | 0.9584 | 0.8969 | 1.2019 |
| Memoria generated | O3 | 0.1973 | 0.0841 | 0.5255 |

**Table 4.2b: Comparisons with the base case**

**Table 4.2: Computation of HELIX on an O2 workstation**

| Cache Type | O2K (origin) | O2 workstation |
|---|---|---|
| Primary (L1) Data Cache | 32KB | 32KB |
| Secondary (L2) Cache | 4MB | 1MB |
| Translation Look aside Buffer | 2MB | 512KB |

**Table 4.3: Cache sizes on our test platforms**

Our study also included computations on CRAY T3E and IBM SP. Execution times for the HELIX code on a single node T3E are shown in Table 4.4. Performances of the original and the Memoria generated code for aggressive compiler option (-O3) are similar, while hand tuned code is about 12% faster. Hand tuned code was about 7% faster than the original code on a single processor SP when compiled with -O3 option.

| Version | OPT=O0 | OPT=O3 |
|---|---|---|
| Original | 12563 | 2038 |
| Hand tuned | 11080 | 1920 |
| Memoria generated | 11814 | 2094 |

**Table 4.4: Computation time (sec) of HELIX on a single node T3E**

## 4. Conclusions

In summary we find the same basic trend in all three platforms, namely hand tuning gave the best results reducing computation time of the original code by 4 to 12%. Memoria performed selective scalar replacement based on the architecture information. Effects of hand coded scalar replacement or scalar replacement by Memoria are more pronounced when no optimization flag is turned on during compilation. Hand tuning, Memoria or a similar tool can help achieve high performance especially when the compiler is not extremely adept in optimization. However, there will be less benefit if the compiler can generate efficient code.

## 4. Acknowledgments

We are grateful to Steve Carr for providing us with the Memoria tool and much advice via email. Thanks to Dave Whalley for many helpful discussions. Steve Bova acted as a contact point for the HELIX code and provided us with a grid file for the HELIX code. This work was supported in part by a grant of HPC time from the DoD HPC Modernization

Program.

## References


Callahan, D., Cocke, J. and Kennedy, K., "Estimating interlock and improving balance for pipelined machines", *Journal of Parallel and Distributed Computing*, Vol 5, pp 334-358, 1988.

Carr, S. and Kennedy, K., "Improving the ratio of memory operations to floating-point operations in loops", *ACM Transactions on Programming Languages and Systems*, pp 1768-1810, Vol. 16, No. 6, 1994a.

Carr, S. and Kennedy, K., "Scalar replacement in the presence of conditional control flow", *Software-Practice and Experience*, pp 51-77, Vol 24(1), 1994b.

Ramachandran, K., Tung, C. and Caradonna, "Rotor Hover Performance Prediction Using a Free-Wake Computational Fluid Dynamics Methods", *Journal of Aircraft*, pp 1105-1110, Vol 26, No 12, 1989.

SpeedShop User's Guide, Document Number 007-3311-003, Silicon Graphics, Inc.

Steinhoff, J. and Ramachandran, K., "Free-wake Analysis of Compressible Rotor Flows", *AIAA Journal*, pp 426-431, Vol 28, No 3, 1990.